Efficient Logic-Based Complex Event Processing and Reactivity Handling

Darko Anicic[1] Paul Fodor[2] Roland Stühmer[1] Nenad Stojanovic[1]

[1] FZI Forschungszentrum Informatik, 76131 Karlsruhe, Germany
[2] State University of New York at Stony Brook, USA

Abstract. In this paper, we present a powerful logic-based language for Complex Event Processing (CEP). CEP deals with finding composed events and has useful applications in areas ranging from agile business and enterprise processes management, financial market applications to active Web-and-service-oriented computation. Many languages for CEP have ad-hoc semantics with unexpected (re)active behavior. Other languages are indeed based on formal logical semantics, however do not have ability to compute complex events in the data-driven (event-driven) fashion. We present a language which features both logic-based semantics and data-driven complex event detection. Moreover, the backward chaining approach allows for very efficient reasoning of complex situations and actions triggered by these events.

1 Introduction

Complex Event Processing (CEP) has the task of processing multiple events with the goal of identifying meaningful event patterns. This goal can further be extended toward detection of particular situations of interest. Detected event patterns and situations are then used to trigger response actions, which in turn may trigger other events. In this cycle, a reactive system is supposed to achieve some useful job. Completeness of the job can be identified either with a particular state being reached (for finite reactive systems), or a set of properties (constraints) being satisfied all the time during the execution (for infinite reactive systems). In both cases (including a mix of them) we have a particular goal or goals that need to be fulfilled. It has been recognized elsewhere [2],[9],[7],[14] that some sort of logic is required to keep event-driven systems running in a controlled manner. The logic should ensure correctness of an event-driven execution. This means that the execution is handled in a way which guarantees satisfiability of predefined constraints on states (in a state-changing environment). Further on, the logic can serve to detect a desired state a user might be interested in; to trace the state of any entity, at any time; to discover conflicting states etc. A number of declarative approaches [7],[14],[9],[2] utilize different kinds of logic to achieve the afore mentioned goals.

However, declarative approaches also have their own limitations. The main common drawback is their inability to do an effective *event or data-driven* computation of event patterns. Data-driven computation means that the occurrence of an event pattern is detected as soon as the last event required for a complete match of the pattern has

occurred. This way of fulfilling goals (i.e., detecting event patterns) is also called *forward chaining*. On the other hand, logic-based approaches are *query or goal-driven*. This means that, for a given event pattern, an event processing engine needs to check if this pattern has been satisfied or not. The check is performed at the time when such a query (goal) is set. Therefore, this way of computation is called query or goal-driven (also known as *backward chaining*). If satisfied by the time when the query is checked, the pattern will be triggered. If not, the pattern cannot be detected until the next time the same check is performed (though it can become satisfied in between the two checks, undetected). This happens as the goal-driven systems try to prove a path *backwards* from a given goal (i.e., event pattern) to the current state (rather than trying to find a path *forward* from the current state that achieves that goal). Backward approaches attempt to *deduce* an event pattern rather than to *detect* it. This is why they have typically been suitable for *request-response* systems. These systems detect an event (or situation) based on a request, followed by an action which is a response. Further on, backward chaining approaches are suitable for periodic detection of event patterns (e.g., each day, every month etc. over historic data). However, they fail when patterns need to be detected as soon as they really occur.

In this paper we present an approach which is based on a *logic*, and still it is *event or data-driven*. This approach specifies complex event patterns declaratively, which allows for powerful logical inference. Since the notions of event patterns, states and response actions are all put in a unified logical framework, the inference process (specifically deduction) can be utilized over all of them. For example, a set of state-changing actions can be triggered by detected event patterns. During the execution they may produce other events, which will be used for detection of other event patterns and so on. The new patterns can further be used to synchronize the ongoing actions; control their execution w.r.t the order in which they are executed; trace the state of any of these actions such that certain constraints are met at all times (otherwise the current state will be rolled back to the last one which meets the constrains), etc.

Our approach is based on decomposition of complex event patterns into *intermediate patterns* (i.e., *goals*). The status of achieved goals at the current state shows the progress toward matching of one or more event patterns. Goals are automatically asserted as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or complete pattern. Important characteristics of these goals are that they are asserted only if they are used later on (to support a more complex goal or an event pattern), goals are all unique, and goals persist as long as they remain relevant (after the relevant period they are deleted). Goals are asserted by declarative rules, which are executed in the backward chaining mode. A specific property of these rules is that they are event-driven.

The paper is organized in the following manner. Section 2 reviews related work and compares it to our work. Section 3 introduces Concurrent Transaction Logic ($\mathcal{CTR}$) as the main underlying formalism of our logic-based approach. In Section **??** we define an event pattern, and explain the basic concepts of event processing based on that definition. Section 6 describes in detail the data-driven event detection, and implements all event operators defined in Section **??**. Section 7 extends event processing with the condition and action parts. Both are implemented within the same formalism as events,

hence enabling a unified reasoning capabilities and full control over a reactive system. Finally, in Section 9 we summarize our work and give an outline for the future work.

2 Related Work

In order to capture relevant changes in a system and respond to those changes adequately, a number of logic-based reactive frameworks have been proposed. This section briefly overviews some approaches for events and reactive rules processing. Work on modeling *behavioral* aspects of an application (using various forms of reactive rules) started in the Active Database community a long time ago. Different aspects have been studied extensively, ranging from modeling and execution of rules to discussing architectural issues [15]. However, what is clearly missing in this work is a clean integration of active behavior with *deductive* and *temporal* capabilities. This is exactly a goal of our approach. Work in [7] goes toward this direction, i.e., putting event patterns in a *logical framework*. As pointed out there, rules can be effectively used for describing, so called, "virtual" event patterns. There exist a number of other reasons to use rules, i.e., rules serve as an abstraction mechanism and offer a higher-level event description. Rules allow for an easy extraction of different views of the same reactive system. Rules are suitable to mediate between the same events differently represented in various interacting reactive systems. Finally, rules can be used for reasoning about causal relationships between events. However in [7], the authors focus only *event queries*, and not on event patterns detectable in a *data-driven* fashion. Discovery of complex events by posing event queries is by no means useless (e.g., querying past data such as log files to detect new events that happened). However it is not sufficient when we talk about Complex Event Processing (CEP). CEP aims to detect complex patterns in *real-time*, i.e., as soon as they occur (and not at the moment when a reactive system poses a query).

In [14], a homogeneous reaction rule language was proposed. From a linguistic standpoint, the approach combines different paradigms such as complex events, reactive rules, declarative rules and integrity constraints. However it suffers from the same drawbacks as mentioned in [7], i.e., complex events are discovered at the moment when corresponding queries are processed, rather than when they have really happened. For instance, the language enables the construction of queries that are processed at given intervals e.g., every 10 seconds trying to discover new events. Clearly, such an approach raises issues about performance and practical usability of the language despite its numerous strengths.

The problem with these and other logic-based approaches is that event processing relies on inference engines with no capability to do event-driven processing. Inference procedures are either based on backward or forward chaining of rules. As mentioned above, backward chaining (or goal-driven reasoning) may be employed to find suitable data (i.e., simple events) answering the question of validity of a given rule (i.e., event pattern). This process is initiated in a top-down manner, starting from a given rule, down to to the simple facts. Forward chaining starts bottom-up with facts available from the database, and aims to deduct all possible truths grounded by a set of rules. Both procedures, as well as a mix of them, cannot detect events by matching the patterns as soon as a relevant fact (event) occur.

Our work contrasts with the afore mentioned approaches by implementing powerful event processing in a *logical manner*, yet enabling their *data-driven* computation.

Close to our approach is [9]. It is an attempt to implement business rules with the Rete Algorithm using subgoals. The work deals with data-driven backward chaining. It has deductive capabilities, and detect fulfilled conditions in business rules (using backward chaining), as soon as relevant facts become available. In our work, we focus rather on complex event detection, and put everything in a unified logical framework covering not only events but conditions and actions (i.e., reactions on events), too.

[2] is an attempt which combines ECA rules with Process Algebra. The idea is to enrich the *action* part, with the declarative semantics of Process Algebra, particularly CCS algebra [13]. The use of the Process Algebra specification aims to enable the *reasoning* capabilities in such an ECA system. Once again, our work differentiates from [2] as it integrates powerful complex event processing with rich conditional capabilities and actions, all in one declarative framework.

Further on, IBM has been developed an event processing tool available in IBM Tivoli Enterprise Console. The engine is capable of processing event patterns and executing ECA rules. The approach uses the Prolog programming language to implement a formalism for specifying event patterns, event filters, and actions. Although the formalism itself is very expressive, the whole approach is rather complex. We aim to provide a powerful, yet simple and intuitive declarative framework for event and action processing.

Concluding this section, all mentioned studies are motivated to use more formal semantics. Our approach based on $\mathcal{CTR}$ may also be seen as an attempt towards that goal, though followed by a pure Logic Programming style. Apart from this, it feature data-driven computation of complex events as well as reach deductive capabilities.

3 CTR Overview

Concurrent Transaction Logic ($\mathcal{CTR}$) [5] is a general logic designed specifically for the rule-based paradigm, and intended to provide a declarative account for state-changing actions. A model and proof theory of $\mathcal{CTR}$ has been defined in [5]. It is an extension of Transaction Logic ($\mathcal{TR}$) [4], with its "Horn" fragment defined. In this section we will informally introduced syntax and semantics of $\mathcal{CTR}$. For further considerations related to the semantics of $\mathcal{CTR}$ the reader is referred to [4].

The atomic formulas of $\mathcal{CTR}$ have the form $p(t_1, ..., t_n)$, where p is a predicate symbol and the t_i's are function terms. The elementary transitions **ins/1** and **del/1** specify basic updates of the current state of the database executed by an oracle that takes the database into a new state by inserting or deleting an atom in the extended database. Complex formulas are built out of atomic formulas and elementary transitions using connectives and quantifiers.

The same as in classical logic, $\mathcal{CTR}$ has $\wedge, \vee, \neg, \forall$, and $\exists$. Unlike in classical logic, $\mathcal{CTR}$ has two connectives, $\otimes$ (serial conjunction) and $|$ (concurrent conjunction); operators: $\Diamond$ (executional possibility), $\Box$ (executional necessity) and $\odot$ (modality of isolation, i.e., isolated execution).

In short, $\mathcal{CTR}$ is a logic for state-changing actions. The truth of $\mathcal{CTR}$ formulas is determined over *paths*. A path is a finite sequence of states. If a formula, ψ, is true over a path $< s_1, ..., s_n >$ it means that ψ can be executed[3] starting with state s_1. During the execution, ψ will change the current state to $s_2, s_3, ...$ and finally terminate at the state s_n. Having this in mind, the intended semantics of $\mathcal{CTR}$ connectives and modal operators can be summarized as follows:

- $\phi \otimes \psi$ means: execute ϕ, then execute ψ;
- $\phi \mid \psi$ means: execute ϕ and ψ concurrently;
- $\phi \wedge \psi$ means: ϕ and ψ must both be executed along the same path;
- $\phi \vee \psi$ means: execute ϕ or ψ nondeterministically;
- $\neg\, \phi$ means: execute in any way, provided that this will not be a valid execution of ϕ;
- $\odot\, \phi$ means: execute ϕ in isolation of other possible concurrently running activities;
- $\Diamond\, \phi$ means: check whether it is possible to execute ϕ at the current state;
- $\Box\, \phi$ means: check necessity to execute ϕ at the current state.

The logic has notions of *data* and *transition* oracles. The *data* oracle, $\mathcal{O}^d(\mathbf{D})$, is used to solve queries related to a particular state $\mathbf{D}$. Likewise, the *transition* oracle, $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, is used to specify an update transition: **ins/1** and **del/1**. If $a \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, then a is an elementary update that changes state $\mathbf{D}_1$ into state $\mathbf{D}_2$. For example, the state change can happen if an atom or a Horn rule is inserted or deleted from $\mathbf{D}$.

For further details about $\mathcal{CTR}$ semantics, its "Horn" fragment etc., the reader is referred to [5], [4], [17].

4 Syntax of CTR Language for Events

The syntax of $\mathcal{CTR} - \mathcal{E}$ is a subset of syntax of $\mathcal{CTR}$, [5]. It includes a set $\mathcal{F}$ of function symbols, a set $\mathcal{V}$ of variables and a set $\mathcal{P}$ of predicate symbols, as well as $\mathcal{CTR}$ connectives $\otimes, \wedge, |, \vee, \neg$ and the quantifier $\forall$. In contrast, the syntax of $\mathcal{CTR} - \mathcal{E}$ does not offer the use of *elementary transitions*, defined in *transition oracle* $\mathcal{O}^t$ (as in [5]). However later in Section 6, we will see that a program created by a user is before execution (at designed time) transformed into a program which indeed contains elementary transitions, **ins/1** and **del/1**. There, **ins/1** is used to insert an event which has just happened, while **del/1** is used to remove unnecessary events from an event history once they cannot be used further, in any pattern.

The simplest formula of $\mathcal{CTR} - \mathcal{E}$ are *atomic formulas* or *atomic events*. An *atomic event* refers to an instantaneous occurrence of interest. Atomic events are used to build *complex events* (event patterns). In general, an event pattern is a pattern built out of (atomic and complex) events, satisfying certain relational operators and data parameters. If a and b are two events, then so are $a \otimes b$ (sequence of events), $a \wedge b$ (classical conjunction), $a|b$ (concurrent conjunction), $a \vee b$ (disjunction of events), $\neg a$ (negated event) and $(\forall X)a$, where X is a variable. A rule is a formula of the form $a \leftarrow b$,

[3] $\mathcal{CTR}$ has a notion of *executional entailment* over that *execution* path, i.e., a logical account of transaction formulas execution [4].

where a is an event, and b is either an atomic or a complex event pattern. In general, every a is defined over a time interval $[T_1, T_2]$ and a set of data terms, i.e., we write $e^{[T_1,T_2]}(X_1, ..., X_n)$. A data term may be a variable, constant or a function symbol.

5 Declarative Semantics

Event history and comparison to $\mathcal{CTR}$ m-paths. The semantics of Concurrent Transaction Logic is based on sequences of paths, called *multi-paths* (*m-paths*) [5]. In general, execution of a transaction changes the database from its initial state to some final state. A finite sequence of states that the database passes along (in order to reach the final state) is called a *path*. To provide the basis for a logical semantics of concurrent conjunction in $\mathcal{CTR}$, the notion of a *path* has been extended to an *m-path* [5]. In Event Processing, the notion of m-path is seen as a *trace of events* that have occurred. We call such a trace an *event multi-path* or the *event history*[4]. An event multi-path (abbr. event m-path) can also be seen as a sequence of *event paths*. In this similarity, there are also fundamental differences between m-paths and event m-paths. An m-path is a finite sequence of paths, where each constituent path represents a period of continuous execution. Further on, constituent paths don't overlap. On the other hand, an event m-path is an *infinite*[5] sequence of event paths, where constituent paths may *overlap*. For instance, detection of two parallel complex events requires detection of their time intervals which must overlap. Moreover, paths can be shared in an event m-path. Again as they can overlap, a piece of a path can be "reused" for detection of other complex events (on other paths). For example, the occurrence of an event that constitutes two other complex events will cause two corresponding paths to share the event history[6].

Operations on Event Paths. Logical operators of Concurrent Transaction Logic are semantically grounded on three operations on m-paths, i.e., *concatenation*, *interleaving* and *reduction*. For their definitions, a reader is referred to [5]. In this paper we extend $\mathcal{CTR}$ to feature capabilities for Complex Event Processing. Therefore we additionally define three operations on paths called *extension*, *chronological extension* and *overlapping*.

A path is a *sequence of states* $(\mathbf{D}_i, \mathbf{D}_{i+1}, \mathbf{D}_{i+2}...)$. A state $\mathbf{D}_i$ changes to its successor $\mathbf{D}_{i+1}$ whenever one or more events occur. This change happens instantaneously, i.e., in time equal to zero. However, there is a time point T_k associated with the change. We assume a (discrete) time model, $(\mathcal{T}, <)$, that can be represented as an ordered set of time points $(...T_{k-1}, T_k, T_{k+1}...)$, where each time point $T \in \mathcal{T}$. For example, we can say that some state change happened at the time point T_k. Points are represented as integers, but other time models for time and data representation are possible with no restriction.

The notion of an event e is characterized by an *event path* (a time interval) π and a set of data parameters (terms), i.e., we write $e^{\pi}(X_1, ..., X_n)$. The event path $\pi = \langle T_i, T_{i+1}, T_{i+2}, ..., T_{i+n} \rangle$ (abbr. $\pi = [T_i, T_{i+n}]$defines a time interval during which the

[4] Recall that an m-path in $\mathcal{CTR}$ records the *execution history of a process*.

[5] In general case, events may never stop to occur, forming unbounded streams.

[6] E.g., provided that in the definition of any of those complex events the use of *negation* has not previously disqualified the detection of the respective other event.

event e has occurred. It is convenient (as in [8]) to define the following simple operations on time intervals: $start([T_1, T_2]) = T_1$, $end([T_1, T_2]) = T_2$, $[T_1, T_2] \cup [T_3, T_4] = [min\{T_1, T_3\}, max\{T_2, T_4\}]$ and $[T_2, T_3] \subseteq [T_1, T_4]$ iff $T_1 \leq T_2$ and $T_3 \leq T_4$. Data terms (i.e. parameters, payload), $X_1, ..., X_n$ of each event, represent data relevant for event-driven applications. A data term may be a variable, constant or a function symbol.

Definition (Extension). If π_1 and π_2 are two event paths. Then there exist paths $\pi_{EXT_1} = \pi_1 \leftrightsquigarrow \pi_2$ and $\pi_{EXT_2} = \pi_2 \leftrightsquigarrow \pi_1$. The path π_{EXT}, that is either π_{EXT_1} or π_{EXT_2}, is an *extension* of π_1 and π_2 if π_{EXT} contains both, π_1 and π_2, and possibly an event path between them (if such a path exists). ■

Definition (Chronological Extension). If π_1 and π_2 are two event paths, where $start(\pi_1) < start(\pi_2)$ and $end(\pi_1) < end(\pi_2)$. Then there exists a path $\pi_{CEXT} = \pi_1 \rightsquigarrow \pi_2$ which is a *chronological extension* of π_1 and π_2 if π_{CEXT} contains π_1 followed by π_2, and possible an event path between them (if such a path exists). ■

It is worth noting that *extension* and *chronological extension* are defined based on event paths containing unknown time points. This feature can be used to detect complex events in situations when we don't know time occurrences of their contributing events (however, we do know they have happened on certain event paths). For instance, among other things the work in [17] shows how to create a set of *event constraints* in $\mathcal{CTR}$. More importantly, it provides logic-based methods for *reasoning* about event constraints. An event constraint is used to ensure that a certain event happens at a certain *state* during some process execution. So it is not important to know when precisely that event has occurred. The emphasis is on a proof that the event has really happened, which suffices for the execution to continue. If the event does not occur, an inference system tries to find another path which is a valid *execution* path in that situation [17]. Hence, the goal of *reasoning* about events and processes is to find a working solution (of course, if such a solution exists). In conclusion, operations on event paths with unknown time points do allow *detection* of complex events. However, they don't allow detection of borders of event paths on which these events happened. This can be very useful for integration of events with processes, as well as reasoning about them.

The *extension* and *chronological extension* operations can also be defined in cases when time points are indeed known. An extended path of $\pi_1[T_i, T_{i+n}]$ and $\pi_2[T_j, T_{j+m}]$ is the path $\pi_{EXT} = \pi_1 \leftrightsquigarrow \pi_2 = \pi_2 \leftrightsquigarrow \pi_1$, defined on an interval $[T_i, T_{i+n}] \cup [T_j, T_{j+m}] = [min\{T_i, T_j\}, max\{T_{i+n}, T_{j+m}\}]$. Similarly $\pi_{CEXT} = \pi_1 \rightsquigarrow \pi_2$ can be defined, too, although the operation $\rightsquigarrow$ is not commutative as $\leftrightsquigarrow$. Additionally the condition, $T_{i+n} < T_j$, must be satisfied.

Definition (Overlapping). If $\pi_1 = \langle T_{i,1}, T_{i,2}, ..., T_{i,n} \rangle$ and $\pi_2 = \langle T_{j,1}, T_{j,2}, ..., T_{j,m} \rangle$ are two paths, then their overlapping is the path $\pi_1 \coloneqq \pi_2 = \langle T_{k,1}, T_{k,2}, ..., T_{k,p} \rangle$. The path $\pi_1 \coloneqq \pi_2$ exists iff $\exists T_{i,l} \in \pi_1$ such that $T_{j,1} < T_{i,l} < T_{j,m}$, or $\exists T_{j,l} \in \pi_2$ such that $T_{i,1} < T_{j,l} < T_{i,n}$. Elements $T_{k,l}$ are defined as follows:

$$T_{k,l} = \begin{cases} T_{i,l} & \text{if } T_{i,l} \in \pi_1, T_{j,l} \notin \pi_2, \\ T_{j,l} & \text{if } T_{i,l} \notin \pi_1, T_{j,l} \in \pi_2, \\ T_{i,l} & \text{if } T_{i,l} \leq T_{j,l}, T_{i,l} \in \pi_1, T_{j,l} \in \pi_2, \\ T_{j,l} & \text{if } T_{i,l} > T_{j,l}, T_{i,l} \in \pi_1, T_{j,l} \in \pi_2. \end{cases}$$

$T_{i,l} = T_{j,l}$ if $T_{i,l}$ and $T_{j,l}$ are two identifiers, pointing to the same time point from $\mathcal{T}$.

We recall the notion of *multi-path structures* from [5] (Definition 3.4) and modify it to *event path structures*. The only difference is that *multi-path structures* have been defined on *m-paths*, while event path structures are defined on *event paths* (see *Declarative Semantics*, above). An event path structure $\mathbf{M}$ over a language $\mathcal{L}$ of $\mathcal{CTR}$ is a triple $\langle U, I_{\mathcal{F}}, I_{path}\rangle$ where:

- U is a set, called the *domain* of $\mathbf{M}$.
- $I_{\mathcal{F}}$ is an interpretation of function symbols in $\mathcal{L}$.
- I_{path} is a mapping from an event path to events. We say that $I_{path}(\pi) \models^{c} e$ if an event, e, has occurred on an event path, π (symbol $\models^{c}$ denotes satisfaction in classical first-order models). Then we also write $I_{path}(\pi) \models^{c} e^{\pi}$.

Since complex event expressions contain variables which have to be bound, we also need to introduce *substitution* (i.e., *variable assignments*). A variable assignment, ν, is a mapping $\nu \mapsto U$, which takes a variable name as input, and returns a domain element as output.

Definition (Satisfaction). Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{path}\rangle$ be an event path structure, let π be an arbitrary event path, let ν be a variable assignment. Then,

1. Base Case: $\mathbf{M}, \pi \models_{\nu} e^{\pi_1}(a_1, ..., a_n)$ iff $I_{path}(\pi) \models^{c}_{\nu} e^{\pi_1}(a_1, ..., a_n)$ and $\pi_1 \subseteq \pi$, for any atomic event $e^{\pi_1}(a_1, ..., a_n)$.
2. Negation: $\mathbf{M}, \pi \models_{\nu} \neg e^{\pi}$ iff it is not the case that $\mathbf{M}, \pi \models_{\nu} e^{\pi}$.
3. Classical Conjunction: $\mathbf{M}, \pi \models_{\nu} e^{\pi_1} \wedge f^{\pi_2}$ iff $\mathbf{M}, \pi_1 \models_{\nu} e^{\pi_1}$ and $\mathbf{M}, \pi_2 \models_{\nu} f^{\pi_2}$, for some event paths π_1, π_2 whose extension $\pi_1 \leftrightsquigarrow \pi_2$ is π. Further on, when borders of π_1, π_2 are known, i.e., $\pi_1 = [T_i, T_{i+n}]$ and $\pi_2 = [T_j, T_{j+m}]$, then $\pi = [T_i, T_{i+n}] \cup [T_j, T_{j+m}]$.
4. Serial Conjunction: $\mathbf{M}, \pi \models_{\nu} e^{\pi_1} \otimes f^{\pi_2}$ iff $\mathbf{M}, \pi_1 \models_{\nu} e^{\pi_1}$ and $\mathbf{M}, \pi_2 \models_{\nu} f^{\pi_2}$, for some event paths π_1, π_2 whose chronological extension $\pi_1 \rightsquigarrow \pi_2$ is π. Further on, when borders of π_1, π_2 are known, i.e., $\pi_1 = [T_i, T_{i+n}]$ and $\pi_2 = [T_j, T_{j+m}]$, and $T_{i+n} < T_j$, then $\pi = [T_i, T_{i+n}] \cup [T_j, T_{j+m}]$.
5. Concurrent Conjunction: $\mathbf{M}, \pi \models_{\nu} e^{\pi_1} | f^{\pi_2}$ iff $\mathbf{M}, \pi_1 \models_{\nu} e^{\pi_1}$ and $\mathbf{M}, \pi_2 \models_{\nu} f^{\pi_2}$, for some event paths π_1, π_2 whose overlapping $\pi_1 \backepsilon \pi_2$ is π.
6. Universal Quantification: $\mathbf{M}, \pi \models_{\nu} \forall X.e^{\pi}$ iff $\mathbf{M}, \pi \models_{\mu} e^{\pi}$, for every variable assignment μ that agrees with ν everywhere except on X.
7. Isolated Event: $\mathbf{M}, \pi \models_{\nu} \odot e^{\pi_1}$ for $\pi_1 \subseteq \pi$ iff there does not exist any f^{π_2} such that $\mathbf{M}, \pi \models_{\nu} f^{\pi_2}$ for $\pi_2 \subseteq \pi$.

If $\mathbf{M}, \pi \models e^{\pi_1}$, then we say that the event e^{π_1} is satisfied (or true) on an event path π in structure $\mathbf{M}$. ■

In practice, it is important to detect a complex event, e^{π_1}, as soon as it occurs on a path π. Hence, we are interested in detecting an event path where $\pi = \pi_1$, rather than $\pi \subset \pi_1$.

Definition (Models). An event path $\mathbf{M}$ is an *event-model* (or simply *model*) of an event expression[7] e^{π_1}, written $\mathbf{M} \models e^{\pi_1}$, iff $\mathbf{M}, \pi \models e^{\pi_1}$ for every event path π, $\pi_1 \subseteq \pi$. An

[7] An event expression is an expression which defines a (complex) event, see Definition *Event Pattern* below.

event path structure is a model of a set of event expressions iff it is a model of every event expression in the set. ■

The execution in CEP applications is driven by events. In order to react to an event, we need first to specify event patterns which we want our application to react to. An *event pattern* is a template which matches certain events. For example, an event pattern matches all orders from customers in response to a discount announcement event. Sometimes the discount announcement event is called an *atomic* event, which is used to build a *complex* event. In general, an event pattern is a pattern which is built out of (atomic and complex) events satisfying certain relational operators and data parameters. Based on Definition *Satisfaction*, we formally define an event pattern in $\mathcal{CTR}-\mathcal{E}$ as follows.

Definition (Event Pattern). 5 An event pattern is a formula of the following form:

- an atomic event;
- $(event_1 \otimes event_2 \otimes ... \otimes event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Sequence of events);
- $(event_1 \wedge event_2 \wedge ... \wedge event_n)$, where $n \geq 0$ and each $event_i$ is an event (Classical conjunction);
- $(event_1 | event_2 | ... | event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Concurrent conjunction);
- $\neg event$, where $event$ is an event pattern (Negation);
- $(event_1 \vee event_2 \vee ... \vee event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Disjunction[8]);
- $\odot event$, where $event$ is an event pattern (Isolation).
- $(event_a \otimes event_1 \otimes ... \otimes event_N \otimes event_b)$, where $N \geq 1$ and $event_a$, $event_i$ and $event_b$ are event patterns (N-times sequence of events);
- $(event_a \otimes event_i \otimes ... \otimes event_b)$, where $i \geq 1$ and $event_a$, $event_i$ and $event_b$ are event patterns (*-Times sequence of events);

A rule is a formula of the form $eventA \leftarrow eventB$, where $eventA$ is an event, and $eventB$ is either an atomic or a complex event pattern ■

In the above definition, every $event_i$ is defined over a time interval $[T_1, T_2]$ with a possible set of data terms that are omitted for space reasons.

In following examples we demonstrate the power of $\mathcal{CTR}-\mathcal{E}$, and give justification for its use in processing event patterns.

Example 3.1 defines a complex event, $checkStatus$, which happens "*if a priceChange event is followed by a stockBuy event*". Additionally, the two events must happen within a certain time frame (i.e., $t < 5$).

Example 5.1.

$checkStatus^{[T_1,T_4]}(X,Y,Z,W) \leftarrow priceChange^{[T_1,T_2]}(X,Y) \otimes stockBuy^{[T_3,T_4]}(Z,Y,W) \otimes (T_4 - T_1 < 5)$.

assuming we have defined the following event patterns[9]:

[8] Recall that disjunction, $event_1 \vee event_2$, can be represented as $\neg(\neg event_1 \wedge \neg event_2)$.

[9] Note that for creating event patterns, we also use a number of built-in predicates with predefined meaning (e.g., "$-$" represents "subtraction" and "$<$" denotes the comparison operation "less then" etc.).

– $priceChange^{[T_i,T_j]}(X,Y)$ is an event pattern, that describes the change in the stock price X (e.g., $\pm 5\%$) of a company Y;
– $stockBuy^{[T_i,T_j]}(Z,W,Y)$ defines a transaction, in which, a buyer Z has bought W amount of stocks from a company Y.

In some cases a user may be interested in analyzing past events. For this purpose, we need a possibility, not only to create data-driven patterns, but also to *query events*. In the following example we ask for all events from the past[10] where the change in stock price was bigger than 10%.

Example 5.2.
$? - priceChange^{[T_i,T_j]}(X,Y) \otimes X > 10.$

It is also possible to describe negated events. For instance, Example 3.3 represents a $notFulfilledOrder$ event that occurs when a customer has made a purchase, but the purchase has not been delivered within a certain time. Therefore, we see that the formalism is also capable of supporting *non-monotonic* features (i.e., the existence of an event, which is defined in absence of other events, may be *retracted* if one of the absent event occurs later[11]. Note that since the event stream is infinite, one should always define a time interval as a scope of a query or a rule. In the Example 3.3 the interval in which we check whether an item has been delivered is $[T_3, T_4]$.

Example 5.3.
$notFulfilledOrder^{[T_1,T_4]}(X) \leftarrow purchased^{[T_1,T_2]}(X) \otimes \neg delivered^{[T_3,T_4]}(X) \otimes (T_4 - T_1 > 3)$.

Example 5.4.
$complexEvent^{[T_1,T_4]}(X) \leftarrow eventA^{[T_1,T_2]}(X) \otimes eventB^{[T_3,T_4]}(X)$.

Example 3.4, demonstrates (as well as previous examples) the use of the sequential composition operator (from Definition *Event Pattern*), i.e., a $complexEvent$ will occur if $eventA$ is followed by $eventB$. If $eventA$ has happened, the system needs to "remember" it, and to raise $complexEvent$ once $eventB$ happens. Detection of event patterns is done by following the event (state-change) path. In the general case, executing a $\mathcal{CTR}$ rule, the system may change the state from $\mathbf{D}_1$ to $\mathbf{D}_n$ (i.e., going through states: $\mathbf{D}_1, \mathbf{D}_2, ..., \mathbf{D}_n$). If the rule describes an event pattern, this state-transition may be seen as the progress towards detection of that event pattern. In this way, if $eventA$ has occurred but $eventB$ has not, the system will *wait*, and raise $complexEvent$ *as soon as* $eventB$ is triggered.

Example 5.5.
$? - stockTransaction^{[T_1,T_2]}(companySymbol_1)^{100}$.

Example 3.5 represents a $N - times$ sequence of events, that triggers when a company stock has been transactioned 100 times (within some interval $[T_1, T_2]$). Note that

[10] In this example, we assume that no events are deleted from an event log. Later, in Section **??**, we talk about *deleting events*, which happen once events are consumed (used for detection of certain complex events). Whether events are kept or removed from a log is matter of particular application.

[11] Nonmonotonicity is related to *querying* for complex events.

one can also use unbounded sequences of events with $* - times$. This operator can be extended to represent expressions that cannot be represented by automata (such as: $a^N \otimes b^N$) and aggregates (such as: sum, min, max, avg).

Finally, the modality of isolation operator $\odot$ is used for defining event patterns with additional constraints. Usually a complex event consists of (atomic or complex) events that satisfy some pre-defined pattern. In this respect, that complex event is not dependent on all events (monitored in an event-driven system or an event cloud), but those that constitutes that particular event. However, the modality of isolation operator allow us to construct a composite event that is, apart from its constituting events, also constrained with other events from the system. For instance, a composite event e^{π}, defined as $e^{\pi}(X,Y) \leftarrow \odot(e_1^{\pi_1}(X) \otimes e_2^{\pi_2}(X,Y) \otimes e_3^{\pi_3}(X,Y))$, will be triggered if $e_1^{\pi_1}, e_2^{\pi_2}$, and $e_3^{\pi_3}$ happen next to each other with no other events in-between. Of course, a time interval for such composite events should be clearly defined as a scope over which events are monitored (e.g., $\pi = [T_i, T_{i+n}]$).

6 Operational Semantics

In Section **??** we have defined a complex event pattern. An event pattern is represented declaratively. This section describes how complex events, described in $\mathcal{CTR}$, can be effectively detected at run-time. Our approach is based on a *goal-directed event-driven* rules. The approach is established on decomposition of complex event patterns into *two-input intermediate events* (i.e., *goals*). The status of achieved goals at the current state shows the progress toward completeness of an event pattern. Goals are automatically asserted by rules as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or pattern. In the following subsection we give more details about implementation of each of event pattern operators defined in Section **??**.

6.1 Sequence of Events

We adopt definition for a sequence of events as a set of events ordered by time. For example, the following rules (1-3) define event patterns based on sequences of events (e.g., $e1$ occurs when an event a is followed by an event b, followed by c). Figure 1 depicts a graph representation of events sequences for $e1$, $e2$ and $e3$ (a node with symbol $\otimes$ denotes a sequence). Additionally it shows that $e1$ is used to trigger both $action1$ and $action2$, while $e2$ and $e3$ trigger $action2$ and $action3$, respectively.

Figure 2 gives a graph representation of $e1$, $e2$ and $e3$ too. It is a variant of Figure 1 where each non-atomic event (denoted with $\otimes$) is a two-input node, representing either an intermediate event or a pattern. The latter figure is more convenient to detect a sequence of events, as it compares events two by two (e.g., $e4$ is triggered whenever a is followed by b, and $e1$ when $e4$ is followed by c).

$$e1([T1,T6]) \leftarrow a([T1,T2]) \otimes b([T3,T4]) \otimes c([T5,T6]). \quad (1)$$
$$e2([T1,T6]) \leftarrow b([T1,T2]) \otimes c([T3,T4]) \otimes d([T5,T6]). \quad (2)$$
$$e3([T1,T4]) \leftarrow c([T1,T2]) \otimes d([T3,T4]). \quad (3)$$

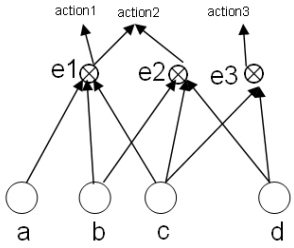

Fig. 1. A sequences of events

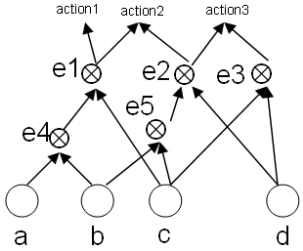

Fig. 2. A left-composed sequence of events

Rules (1)-(3) specify complex event patterns $e1$, $e2$ and $e3$. In general, they are created by a user using the operators from Definition 5 (though currently we are focusing only on sequential composition). In their present form, these rules are not convenient to be used for event-driven computation. These are rather a Prolog-style rules suitable for backward chaining evaluation. Such rules are understood as goals which at certain time either can or cannot be proved by an inference engine. A provable goal means that a corresponding pattern has occurred, and vice versa. Note that backward chaining evaluation is not appropriate for event-driven computation as it does not detect events when they really occur. Rather it *proves* them at the time when corresponding goals (queries) are set. This is why this kind of event processing is called *query-driven*, Section 1.

$$a([T_1,T_2]) : -while\_do(a,1,[T_1,T_2]). \quad (4)$$

$$a(1,[T_1,T_2]) :- ins(goal(b([\_,\_]),a([T_1,T_2]),e4([\_,\_]))). \tag{5}$$

$$b([T_1,T_2]) :- while\_do(b,1,[T_1,T_2]). \tag{6}$$

$$\begin{array}{l} b(1,[T_3,T_4]) :- goal(b([T_3,T_4]),a([T_1,T_2]),e4([\_,\_]))\otimes \\ del(goal(b([T_3,T_4]),a([T_1,T_2]),e4([\_,\_]))) \otimes e4([T_1,T_4]). \end{array} \tag{7}$$

$$b(2,[T_1,T_2]) :- ins(goal(c([\_,\_]),b([T_1,T_2]),e5([\_,\_]))). \tag{8}$$

$$\begin{array}{l} c([T_1,T_2]) :- while\_do(c,1,[T_1,T_2]). \\ c(1,[T_3,T_4]) :- goal(c([T_3,T_4]),e4([T_1,T_2]),e1([\_,\_]))\otimes \\ del(goal(c([T_3,T_4]),e4([T_1,T_2]),e1([\_,\_]))) \otimes e1([T_1,T_4]). \\ c(2,[T_3,T_4]) :- goal(c([T_3,T_4]),b([T_1,T_2]),e5([\_,\_]))\otimes \\ del(goal(c([T_3,T_4]),b([T_1,T_2]),e5([\_,\_]))) \otimes e5([T_1,T_4]). \\ c(3,[T_1,T_2]) :- ins(goal(d([\_,\_]),c([T_1,T_2]),e3([\_,\_]))). \end{array} \tag{9}$$

$$\begin{array}{l} d([T_1,T_2]) :- while\_do(d,1,[T_1,T_2]). \\ d(1,[T_3,T_4]) :- goal(c([T_3,T_4]),e5([T_1,T_2]),e2([\_,\_]))\otimes \\ del(goal(c([T_3,T_4]),e5([T_1,T_2]),e2([\_,\_]))) \otimes e2([T_1,T_4]). \\ d(2,[T_3,T_4]) :- goal(d([T_3,T_4]),c([T_1,T_2]),e3([\_,\_]))\otimes \\ del(goal(d([T_3,T_4]),c([T_1,T_2]),e3([\_,\_]))) \otimes e3([T_1,T_4]). \end{array} \tag{10}$$

$$\begin{array}{l} e4([T_1,T_2]) : -while\_do(e4,1,[T_1,T_2]). \\ e4(1,[T_1,T_2]) : -ins(goal(c([\_,\_]),e4([T_1,T_2]),e1([\_,\_]))). \end{array} \tag{11}$$

$$\begin{array}{l} e5([T_1,T_2]) : -while\_do(e5,1,[T_1,T_2]). \\ e5(1,[T_1,T_2]) : -ins(goal(c([\_,\_]),e5([T_1,T_2]),e2([\_,\_]))). \end{array} \tag{12}$$

$$\begin{array}{l} e1([T_1,T_2]) :- while\_do(e1,1,[T_1,T_2]). \\ e1(1,[T_1,T_2]) :- action1([T_1,T_2]). \\ e1(2,[T_1,T_2]) :- action2([T_1,T_2]). \end{array} \tag{13}$$

$$\begin{array}{l} e2([T_1,T_2]) : -while\_do(e2,1,[T_1,T_2]). \\ e2(1,[T_1,T_2]) : -action2([T_1,T_2]). \end{array} \tag{14}$$

$$\begin{array}{l} e3([T_1,T_2]) : -while_do(e3,1,[T_1,T_2]). \\ e3(1,[T_1,T_2]) : -action3([T_1,T_2]). \end{array} \tag{15}$$

$$\begin{array}{l} action1([T_1,T_2]) : -write(action1([T_1,T_2])),nl. \\ action2([T_1,T_2]) : -write(action2([T_1,T_2])),nl. \\ action3([T_1,T_2]) : -write(action3([T_1,T_2])),nl. \end{array} \tag{16}$$

$$\begin{array}{l} while\_do(Pred,N,L) : -(FullPred = ..[Pred,N,L]) \\ \otimes execCTR(FullPred)\otimes \\ (N1 is N+1) \otimes while\_do(Pred,N1,L). \\ while\_do(Pred,N,L) : -true. \end{array} \tag{17}$$

Rules (4)-(12) represent our example rules (1)-(3), rewritten so to enable *event-driven backward chaining*. First, they are suitable for detection of events as soon as they occur. Second, detection of events is focused on patterns of interest, rather than merely detecting irrelevant events. Each rule from (4)-(12) falls in one of two types of

rules. One type is used to *generate goals*, and another one generates either intermediate events (e.g., $e4$, $e5$) or pattern events (e.g., $e1$, $e2$), i.e., *check rules*.

The first type of rules are used to *generate goals* (i.e., insert goals into database). Every generated goal represents that an event has occurred. For example, rule (5) inserts $goal(b([\_,\_]), a([T_1,T_2]), e4([\_,\_])))$ in the database every time an event of type a occurs (utilizing ins construct from $\mathcal{CTR}$). A goal consists of three events (terms). Its interpretation is that "an instance of event a has occurred at $[T_1,T_2]$[12], and we are waiting for b to happen in order to detect $e4$", (Figure 2). Obviously the goal does not carry information about times for b and $e4$, as we don't know when they will occur. In general, the second event in a goal always denotes an event that has just occurred. The role of the first event is to specify what we are waiting for to detect an event that is on the third position. In our example, if a was used to build one more pattern (apart from the pattern $e4$), we would have an additional rule which inserts an additional goal. For instance, $goal(b([\_,\_]), a([T_1,T_2]), e5([\_,\_])))$ could be used to detect $e5$ by the same sequence of a and b. In general, the number of rules that insert goals depend on number of *left-to-right* parent edges (see Figure 3(a)).

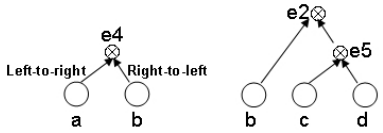

Fig. 3. A binary event goal (a). A right-composed sequence of events (b)

The second type of rules are *check rules*. checks whether a certain goal already exists in the database, in which case it triggers an event. For example, rule (7) will fire whenever b occurs. The rule checks whether
$goal(b([T_3,T_4]), a([T_1,T_2]), e4([\_,\_]))$ already exists (i.e., a previously has happened), in which case it triggers $e4$ (by calling $e4([T_1,T_4])$. The time occurrence of $e4$ (i.e., $[T_1,T_4]$) is defined based on the occurrence of constituting events (i.e., $a([T_1,T_2])$, and $b([T_3,T_4])$). Calling $e4([T_1,T_4])$, this event is effectively propagated either upward (if it is an intermediate event) or used for triggering an action (if it is event pattern, e.g., $e1$).

We see that, in general with backward chaining, goals are crucial for computation of complex events. They show the current state of progress toward matching an event pattern. Moreover they allow for detection of the state of any complex event, at any time. Therefore goals can enable *reasoning* over events, e.g., *what are events missing in order to detect some a certain complex event pattern*. They can persist over a period of time. It is worth noting that rules of the second type (e.g., rule (7)) also delete goals (using del construct featured by $\mathcal{CTR}$) [5]). Once a goal is "consumed", it is removed

[12] Apart from the time stamp, an event may carry other data parameters that are omitted here in order to make examples more readable.

from the database[13]. In this way goals are kept persisted as long as (but not longer) they are needed.

Rules (13)-(15) represent complex events used for triggering a set of actions. As shown in Figure 1, $e1$ triggers $action1$ and $action2$, $e2$ and $e3$ activate $action2$ and $action3$, respectively. Further on, rules (16) define what tasks each of actions needs to execute. In our example, they only print its name with the time when an action has been triggered. In general case, an action may be more complex. For instance, it can be composed of more concurrently running sub-actions which are synchronized by detected complex events and/or between themselves. More details about actions are provided in Section 7. Finally rules (17) implements a while-do loop, which for any occurrence of an event goes through each rule specified for that event. For example, when c occurs the first rule in the set of rules (9) will be fired. This rule will then loop invoking all other rules specified for c (those with c in the rule head), e.g., $c(1)$, $c(2)$ and $c(3)$.

Figure 4 shows a listing after the execution of rules (4)-(17). We have assumed a situation where a sequence of atomic events of type a, b, c, d has occurred in time points $T_i = 1, 2, 3, 4$[14]. As a result, when c occurred (i.e., at $T_i = 3$), $action1$ and $action3$ have been triggered (i.e., with a detection interval $[1, 3]$ for $e1$). Similarly after occurrence of d, $action1$ and $action3$ have been activated as consequences of detection $e2$ and $e3$ (that they only have different intervals as the time of detection for $e2$ and $e3$ is different too).

In the following we will briefly explain how rules (4)-(17), for the given sequence, compute $e1$, $e2$, $e3$[15]. First, when an instance of a occurs rule (4) will fire and run a loop (i.e., rule (17)). In case of event a there is only one additional rule (i.e., rule (5)) to be executed in that loop. This rule will insert
$goal(b([\_, \_]), a([T_1, T_2]), e4([\_, \_]))$. Let us assume that some time after, b occurs (e.g., in $T_1 = 2$). Similarly this event will fire rule (6), which in turn will fire rules (7) and (8). The first rule will check whether
$goal(b([\_, \_]), a([T_1, T_2]), e4([\_, \_]))$ exists in the database (i.e., a has previously happened), in which case it will trigger $e4([T_1, T_4])$. The second rule will insert
$goal(c([\_, \_]), b([T_1, T_2]), e5([\_, \_]))$ denoting that eventual occurrence of c would cause $e5$ to happen. By calling $e4$ from rule (7), this event is effectively propagated upward. A goal (i.e.,
$goal(c([\_, \_]), e4([T_1, T_2]), e1([\_, \_])))$ will be inserted into the database, meaning that an occurrence of c will trigger $e1$ (see the second rule in rules (11)). Now when c occurs, rules (9) will all fire. First, it will be checked whether $e4$ has happened, what is required to trigger $e1([1, 3])$ (i.e., final complex event). Second, a proof that b has happened will trigger $e5$, which further sets a goal waiting for d. Third,
$goal(d([\_, \_]), c([T_1, T_2]), e3([\_, \_]))$ will be asserted. Finally when d occurs, it will fire both complex events $e2([2, 4])$ and $e3([3, 4])$.

---

[13] In situation where past events are useful for further computation the deletion operation, of course, can be omitted.

[14] $T_1 = T_2$ holds for any atomic event $a([T_1, T_2])$.

[15] By computing $e1$, $e2$, $e3$, we mean more precisely computation of *instances* of $e1$, $e2$, $e3$. The same remark applies in the rest of this paragraph.

```
xsb -e "[load],init,ctr_comp('event_02')."
yes
| ?- execCTR(a([1,1])).

yes
| ?- execCTR(b([2,2])).

yes
| ?- execCTR(c([3,3])).
action1([1,3])
action2([1,3])

yes
| ?- execCTR(d([4,4])).
action2([2,4])
action3([3,4])

yes
| ?-
```

Fig. 4. Detection of sequence of events and action triggering

Since in our running example events $e1$, $e2$ and $e3$ have all been detected, there is no need to keep intermediate events (i.e., $e4$ and $e5$) in the database. They have been used for detection of complex events at different moments during the interval $[1, 4]$, and deleted as soon as they have become irrelevant. A query from Figure 5 proofs this out.

```
| ?- goal(_,_,_).

no
| ?-
```

Fig. 5. Non relevant goals are not kept in memory

Now when we have explained how rules (4)-(17) compute complex events in a data-driven backward chaining fashion, let us provide more details on structure of these rules and goals. They are designed so to allow an *transformation* from user defined rules (e.g., rules (1)-(3)), and enable efficient event pattern detection. Recall that there exist two types of rules, i.e., one used to generate goals, and another one to trigger events. For any occurrence of an event in the body of user defined rules, one of those two kinds of rules will be added (in the rule base during the transformation). Apart from this, for every particular event (predicate) there will be added one additional rule to define a while-do loop for that event. For instance, there is only one occurrence of a in rules (1)-(3), hence there is only one rule added (in a set of rules (4)-(12)) for event a, and one to define a loop. Further on, c has occurred three times, thus we have $c(1, [\_, \_])$, $c(2, [\_, \_])$ and $c(3, [\_, \_])$ as well as one rule for a loop in rules (9). In general, for a given set of rules defining complex patterns there will be as many transformed rules as usages of any particular atomic event. The set of transformed rules is further accompanied with rules for loops (as many as there are distinguished atomic events). Finally the same procedure for generating transformed rules is repeated for intermediate events (e.g., $e4$,

$e5$). The complete transformation is proportional to the number of user defined event pattern rules, hence such a transformation can be linear at the design time.

In the following, we give more details about computation of intermediate events and goals, which essentially enable the linear transformation. We have already seen that Figure 2 depicts intermediate events by representing them as two-input nodes. An intermediate event is built as a composition of *binary events*. For example, a sequence of events $a \otimes b \otimes c \otimes d$ can be presented as a sequence of binary events $(((a \otimes b) \otimes c) \otimes d)$. This would produce the following intermediate events $ie_1 \leftarrow a \otimes b$, $ie_2 \leftarrow ie_1 \otimes c$ and $ie_3 \leftarrow ie2 \otimes d$. To effectively handle detection of intermediate events and thus complex patterns, transformed rules (e.g., (4)-(12)) manipulate with insertion and deletion of goals in the database. Note that similarly as an intermediate event, each goal (e.g., $goal(b, a, e4)$) constitutes one two-input node, see Figure 2. Two inputs are labeled with a left-hand side event, i.e., an event at the second position in the goal (e.g., a in $goal(b, a, e4)$), and a right-hand side event, i.e., an event positioned as the first one (e.g., b). The node itself is labeled with an event at the third position (e.g., $e4$), Figure 3(a). Two input nodes can be any events, i.e., atomic events, intermediate events or event patterns. They are constructed in a direction *left-to-right* (i.e., a *left-composed* events). The reason for this is to avoid *negated goals* (when constructing sequences of events). Figure 3(b) represents a *right-composed* sequence for $e2$ (from Figure 2), rules (18-21) are created so to implement this sequence. We see that rule 18, unlike its equivalent rule 8 (from Figure 2), has a negated goal. If this goal did not exist both sequences b, c and d as well as c, b, d would be valid (which is obviously incorrect). In order to avoid unnecessary negated goals we adhere to left-composed events.

$$b(1,[T_1,T_2]) :- not(goal(d([\_,\_]), c([T_1,T_2]), e5([\_,\_]))) \otimes ins(goal(e5([\_,\_]), b([T_1,T_2]), e2([\_,\_]))). \quad (18)$$

$$c(1,[T_1,T_2]) :- ins(goal(d([\_,\_]), c([T_1,T_2]), e5([\_,\_]))). \quad (19)$$

$$d(1,[T_3,T_4]) :- goal(d([\_,\_]), c([T_1,T_2]), e5([\_,\_])) \otimes del(goal(d([T_3,T_4]), c([T_1,T_2]), e5([\_,\_]))) \otimes e5([T_1,T_4]). \quad (20)$$

$$e5(1,[T_3,T_4]) : -goal(e5([\_,\_]), b([T_1,T_2]), e2([\_,\_])) \otimes del(goal(e5([\_,\_]), b([T_1,T_2]), e2([\_,\_]))) \otimes e2([T_1,T_4]). \quad (21)$$

Finally we sketch an algorithm that automatically transform user defined event pattern rules for *sequential conjunction* into rules which can be used for detection of complex events (using *data-driven backward chaining*), (Algorithm 6.1). First, complex event rules are divided into *binary event goals* (see Figure 2 and Figure 3(a)). Effectively for each pair (e.g., a, b) of events, there is an (intermediate) event created (e.g., $e_1, e_2, ...$). Then for each event binary goal, two kinds of rules are inserted. For the left-hand node (e.g., a), rules that *generate goals* are inserted (e.g., $ins(goal(b^{[\_,\_]}, a^{[T_1,T_2]}, e1^{[\_,\_]}))$). Similarly, for the right-hand node (e.g., b), rules that *check goals* are inserted (e.g., $b^{[T_3,T_4]}(1) : -goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, e1) \otimes del(goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, e1^{[\_,\_]})) \otimes e1^{[T_1,T_4]}$.). Here, rules that check goals are assumed to delete consumed events (i.e., $del(goal(...))$). This feature is optional. Hence the update transition del can be omitted (if consumed events are kept for further processing or analyzing).

Algorithm 6.1 Sequential conjunction.
Divide complex event rules into binary event goals (e.g., $a \otimes b$) by introducing temporary event symbols (e.g., $e1, ...$);
For each event binary goal $e1 : -a \otimes b$. $\{$
insert rules to generate goals:
$a : -while\_do(a, 1)$.
$a^{[T_1,T_2]}(1) : -ins(goal(b^{[.,.]}, a^{[T_1,T_2]}, e1^{[.,.]}))$.
insert check rules:
$b : -while\_do(b, 1)$.
$b^{[T_3,T_4]}(1) : -goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, e1) \otimes del(goal(b^{[T_3,T_4]}, a^{[T_1,T_2]}, e1^{[.,.]})) \otimes e1^{[T_1,T_4]}$.
$\}$

6.2 Classical Conjunction of Events

Classical conjunction is another typical operator in event processing. An event pattern based on a classical conjunction occurs when all events which consist that conjunction occur. Unlike the sequence operator, here the constitutive events can happen at different times with no particular order between them. For example $e1$ in rules (22) is defined as a conjunction of two sequences $e2$ and $e3$.

$$\begin{array}{l} e1([T_5, T_6]) \leftarrow e2([T_1, T_2]) \wedge e3([T_3, T_4]). \\ e2([T_1, T_4]) \leftarrow a([T_1, T_2]) \otimes b([T_3, T_4]). \\ e3([T_1, T_4]) \leftarrow c([T_1, T_2]) \otimes d([T_3, T_4]). \end{array} \quad (22)$$

$e1$ is detected when both events $e2$ and $e3$ occur (i.e., either $e2$ followed by $e3$, or $e3$ followed by $e2$). An interval in which $e1$ is detected (for the given example) is defined by $T_5 = min(T_1, T_3)$ and $T_6 = max(T_2, T_4)$. A corresponding graph for rules (22) is shown in Figure 6.

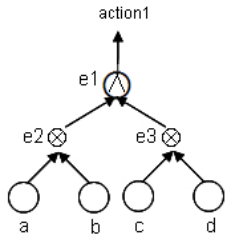

Fig. 6. Classical conjunction of events

Rules (23)-(30) represent rules transformed so to enable detection of conjunctional patterns in an event-driven backward chaining fashion. These rules are created following the same principles (from Subsection 6.1) regarding the generation of goals as a mechanism for event detection. Therefore we will only briefly describe them, taking into account only differences between them and rules (4-17), see Subsection6.1). Due

to space limitation, we will also follow this approach in describing the remaining event pattern operators throughout this section.

The role of rules (23-27) is absolutely the same as in Subsection 6.1, i.e., to detect two sequences of events. Let us assume that the sequence $e2$ was detected as the first one (followed by $e3$ afterwards). In this situation, rules (28) will fire. The loop will execute two rules. The first rule will check whether $e3$ has previously happened, in which case it will trigger $e1([T_5, T_6])$, provided that $T_5 = min(T_1, T_3)$ and $T_5 = max(T_2, T_4)$. Let us repeat once again that the "consumed" goal is removed from the database once it becomes irrelevant. However if $e3$ had not previously happened (by the time when $e2$ occurred), the second rule will insert $goal(e3([\_, \_]), e2([T_3, T_4]), e1([\_, \_]))$. This goal has the meaning that $e2$ occurred in $[T_3, T_4]$, and now we are waiting for $e3$ to happen in order to detect $e1$. Once $e3$ occurs, rules (29) will execute. Again there is a loop firing two rules. The first rule will trigger $e1$ provided that $e2$ has already happened. A time interval on which $e1$ has been detected is the same, i.e., $T_5 = min(T_1, T_3)$ and $T_6 = max(T_2, T_4)$. If $e2$ has not happened, a goal stating that $e3$ happened (in scope of detection of $e1$) is inserted. It is worth noting that a non-occurrence of an event is checked by using a negated goal. In Subsection 6.1 we have discussed left and right-composed nodes in order to avoid use of negated goals. As we see, for the classical conjunction operator we still need negated goals. However their use for this operator is not an issue. Here we can easy form a negated goal as it consists of the same (three) events from its a non-negated counter-goal (see the second and third rule in rules (28)). However in case of right-composed nodes of a sequence of events, an algorithm for the rule transformation would need to add one negated goal to an existing rule every time it encounters a new node (in that sequence). That is, the algorithm would need to continuously modify already created transformation rules as long as the transformation takes place. AS shown, this can be avoided by using left-composed nodes in a sequence of events.

$$\begin{array}{l} a([T_1, T_2]) : -while\_do(a, 1, [T_1, T_2]). \\ a(1, [T_1, T_2]) : -ins(goal(b([\_, \_]), a([T_1, T_2]), e2([\_, \_]))). \end{array} \tag{23}$$

$$\begin{array}{l} b([T_1, T_2]) : - \ while\_do(b, 1, [T_1, T_2]). \\ b(1, [T_3, T_4]) : - \ goal(b([\_, \_]), a([T_1, T_2]), e2([\_, \_])) \otimes \\ del(goal(b([\_, \_]), a([T_1, T_2]), e2([\_, \_]))) \otimes e2([T_1, T_4]). \end{array} \tag{24}$$

$$\begin{array}{l} c([T_1, T_2]) : - \ while\_do(c, 1, [T_1, T_2]). \\ c(1, [T_1, T_2]) : - \ ins(goal(d([\_, \_]), c([T_1, T_2]), e3([\_, \_]))). \end{array} \tag{25}$$

$$\begin{array}{l} d([T_1, T_2]) : - \ while\_do(d, 1, [T_1, T_2]). \\ d(1, [T_3, T_4]) : - \ goal(d([\_, \_]), c([T_1, T_2]), e3([\_, \_])) \otimes \\ del(goal(d([\_, \_]), c([T_1, T_2]), e3([\_, \_]))) \otimes e3([T_1, T_4]). \end{array} \tag{26}$$

$$\begin{array}{l} e1([T_1, T_2]) : -while\_do(e1, 1, [T_1, T_2]). \\ e1(1, [T_1, T_2]) : -action1([T_1, T_2]). \end{array} \tag{27}$$

$$
\begin{array}{l}
e2([T_1,T_2]) : -while\_do(e2,1,[T_1,T_2]).\\
e2(1,[T_3,T_4]) : -goal(e2([\_,\_]),e3([T_1,T_2]),e1([\_,\_]))\otimes\\
del(goal(e2([\_,\_]),e3([T_1,T_2]),e1([\_,\_]))) \otimes min(T_1,T_3,T_5)\otimes\\
max(T_2,T_4,T_6) \otimes e1([T_5,T_6]).\\
e2(1,[T_3,T_4]) : -not(goal(e2([\_,\_]),e3([T_1,T_2]),e1([\_,\_])))\otimes\\
ins(goal(e3([\_,\_]),e2([T_3,T_4]),e1([\_,\_]))).
\end{array}
\tag{28}
$$

$$
\begin{array}{l}
e3([T_1,T_2]) : -while\_do(e3,1,[T_1,T_2]).\\
e3(1,[T_3,T_4]) : -goal(e3([\_,\_]),e2([T_1,T_2]),e1([\_,\_]))\otimes\\
del(goal(e3([\_,\_]),e2([T_1,T_2]),e1([\_,\_]))) \otimes min(T_1,T_3,T_5)\\
max(T_2,T_4,T_6) \otimes e1([T_5,T_6]).\\
e3(1,[T_3,T_4]) : -not(goal(e3([\_,\_]),e2([T_1,T_2]),e1([\_,\_])))\otimes\\
ins(goal(e2([\_,\_]),e3([T_3,T_4]),e1([\_,\_]))).
\end{array}
\tag{29}
$$

$$
\begin{array}{l}
min(T_1,T_2,T_3) : -(T_1 < T_2 \rightarrow T_3 = T_1; T_3 = T_2).\\
max(T_1,T_2,T_3) : -(T_1 > T_2 \rightarrow T_3 = T_1; T_3 = T_2).
\end{array}
\tag{30}
$$

Algorithm 6.2 transforms automatically user defined complex event rules for *classical conjunction* into rules which can be used for detection of complex events (using *data-driven backward chaining*). This means that rules (22), using this algorithm, are transformed into rules (23-29). The procedure for dividing complex event rules into *binary event goals* is the same as in Algorithm 6.1. However rules for *generating* and *checking* goals are different. We have a pair of these rules created for both, a left-hand node (e.g., a) as well as for a right-hand node (e.g., b). When an event that is a left-hand node of some event goal occurs, the algorithm checks whether an event that is a right-hand node of the same goal previously happened. If yes, the time interval (e.g., $[T_5,T_6]$) of an intermediate event (e.g., e_1) is calculated and that event is being triggered (e.g., a call of $e1^{[T_5,T_6]}$). Otherwise, a goal that states that one left-node event has occurred is inserted (e.g., $ins(goal(b^{[\_,\_]},a^{[T_1,T_2]},e1^{[\_,\_]}))$). Again, as in Algorithm 6.1, all deletions of consumed events are optional.

Algorithm 6.2 Classical conjunction.

Divide complex event rules into binary event rules (e.g., $a \wedge b$) by introducing temporary event symbols (e.g., $e1, ...$);

For each event binary goal $e1 : -a \wedge b$. {

insert rules for the left-hand node (i.e., a):

$a : -while\_do(a,1).$

$a^{[T_3,T_4]}(1) : -goal(a^{[\_,\_]},b^{[T_1,T_2]},e1^{[\_,\_]}) \otimes del(goal(a^{[\_,\_]},b^{[T_1,T_2]},e1^{[\_,\_]})) \otimes T_5 = min\{T_1,T_3\} \otimes T_6 = max\{T_2,T_4\} \otimes e1^{[T_5,T_6]}.$

$a^{[T_1,T_2]}(1) : -\neg(goal(a^{[\_,\_]},b^{[T_1,T_2]},e1^{[\_,\_]})) \otimes ins(goal(b^{[\_,\_]},a^{[T_1,T_2]},e1^{[\_,\_]})).$

insert rules for the right-hand node (i.e., b):

$b : -while\_do(b,1).$

$b^{[T_3,T_4]}(1) : -goal(b^{[\_,\_]},a^{[T_1,T_2]},e1^{[\_,\_]}) \otimes del(goal(b^{[\_,\_]},a^{[T_1,T_2]},e1^{[\_,\_]})) \otimes T_5 = min\{T_1,T_3\} \otimes T_6 = max\{T_2,T_4\} \otimes e1^{[T_5,T_6]}.$

$b^{[T_1,T_2]}(1) : -\neg(goal(b^{[\_,\_]},a^{[T_1,T_2]},e1^{[\_,\_]})) \otimes ins(goal(a^{[\_,\_]},b^{[T_1,T_2]},e1^{[\_,\_]})).$

}

Finally rules (23-30) can be executed. By executing them we have obtained a listing presented in Figure 7. For a given set of atomic events, there are six different situations in which classical conjunction of events can be detected[16] (a,b,c,d, then a,c,b,d, etc.). Obviously $e1$ could also have been defined with non-atomic events, in which case we would get different time intervals from these presented in Figure 7.

```
xsb -e "[load],init,ctr_comp('concurrent_conj')."
?- execCTR(a([1,1])),execCTR(b([2,2])),execCTR(c([3,3])),execCTR(d([4,4])).
action1([1,4])

?- execCTR(a([1,1])),execCTR(c([2,2])),execCTR(b([3,3])),execCTR(d([4,4])).
action1 ([1,4])

?- execCTR(a([1,1])),execCTR(c([2,2])),execCTR(d([3,3])),execCTR(b([4,4])).
action1 ([1,4])

?- execCTR(c([1,1])),execCTR(a([2,2])),execCTR(b([3,3])),execCTR(d([4,4])).
action1 ([1,4])

?- execCTR(c([1,1])),execCTR(a([2,2])),execCTR(d([3,3])),execCTR(b([4,4])).
action1 ([1,4])

?- execCTR(c([1,1])),execCTR(d([2,2])),execCTR(a([3,3])),execCTR(b([4,4])).
action1 ([1,4])
```

Fig. 7. Classical conjunction of events - execution

6.3 Concurrent Conjunction of Events

Concurrent conjunction of events can be obtained by slight modification of the classical conjunction operator (Subsection 6.2). A concurrent conjunction of two events is detected when their intervals overlap. For example, $e1$ (rules 31) is detected if events $e2$, and $e3$ both occur and interval in which they are detected overlap.

$$\begin{array}{l} e1 \leftarrow e2|e3. \\ e2([T_1,T_4]) \leftarrow a([T_1,T_2]) \otimes b([T_3,T_4]). \\ e3([T_1,T_4]) \leftarrow c([T_1,T_2]) \otimes d([T_3,T_4]). \end{array} \tag{31}$$

Rules (31) are graphically represented in Figure 8. The operator $|$ in the $e1$ node denotes concurrent conjunction operator. As classical and concurrent conjunction are similar operators, to detect $e1$ we can still use rules ((23)-(30)) from Subsection 6.2. However we need to introduce a condition that T_3 must strictly be less than T_2 (i.e., $T_3 < T_2$) in rules (28) and (29). At the same time, built-in predicates for finding min and max times in both rules become unnecessary (hence they can be removed). $e1$ will be defined on an interval $[T_1, T_4]$ (instead of $[T_5, T_6]$). The *less* condition ensures that intervals of $e2$ and $e3$ overlap. Note that, due to that condition, a concurrent conjunction of events can be built only with non-atomic events.

Algorithm 6.3 automatically transforms user complex event rules for *concurrent conjunction* into rules which serve for computation of complex events in a *data-driven backward chaining* fashion. The procedure for dividing complex event rules into *binary*

[16] It is assumed that $e1$ triggers $action1$.

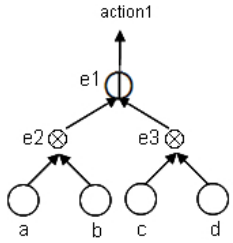

Fig. 8. Concurrent conjunction of events

event goals is the same as in Algorithm 6.1 and Algorithm 6.2. Rules for *generating* and *checking* goals are similar as those in Algorithm 6.2. Additionally in Algorithm 6.3 there is a sufficient condition which ensures the time interval overlapping (i.e., $T_3 < T_2$).

Algorithm 6.3 Concurrent conjunction.
Divide complex event rules into binary event rules (e.g., $a|b$) by introducing temporary event symbols (e.g., $e1, ...$);
For each event binary goal $e1 :- a|b$. {
insert rules for the left-hand node (i.e., a):
$a :- while\_do(a, 1)$.
$a^{[T_3,T_4]}(1) \ : \ -goal(a^{[.,.]}, b^{[T_1,T_2]}, e1^{[.,.]}) \otimes del(goal(a^{[.,.]}, b^{[T_1,T_2]}, e1^{[.,.]})) \otimes T_3 < T_2 \otimes T_5 = min\{T_1, T_3\} \otimes T_6 = max\{T_2, T_4\} \otimes e1^{[T_5,T_6]}$.
$a^{[T_1,T_2]}(1) : -\neg(goal(a^{[.,.]}, b^{[T_1,T_2]}, e1^{[.,.]})) \otimes ins(goal(b^{[.,.]}, a^{[T_1,T_2]}, e1^{[.,.]}))$.
insert rules for the right-hand node (i.e., b):
$b :- while\_do(b, 1)$.
$b^{[T_3,T_4]}(1) \ : \ -goal(b^{[.,.]}, a^{[T_1,T_2]}, e1^{[.,.]}) \otimes del(goal(b^{[.,.]}, a^{[T_1,T_2]}, e1^{[.,.]})) \otimes T_3 < T_2 \otimes T_5 = min\{T_1, T_3\} \otimes T_6 = max\{T_2, T_4\} \otimes e1^{[T_5,T_6]}$.
$b^{[T_1,T_2]}(1) : -\neg(goal(b^{[.,.]}, a^{[T_1,T_2]}, e1^{[.,.]})) \otimes ins(goal(a^{[.,.]}, b^{[T_1,T_2]}, e1^{[.,.]}))$.
}

Executing rules (31) (with rules (23)-(30) modified to meet requirements for the concurrent conjunction operator) and using the same sequences of atomic events from Listing 7, we get four different situations in which $e1$ has bee detected. The first and last situation from Listing 7, in this case, wont be detected as the overlap condition is not satisfied.

6.4 Disjunction of Events

Disjunction of events is detected when any of events that constitutes that disjunction occurs. For instance, rules (32) define $e1$ as a disjunction of two complex events, i.e., $e2$ and $e3$. An interval in which $e1$ has been defined depends on an event which cased $e1$ to happen. In our example, the interval is either taken from $e2$ or from $e3$.

$$
\begin{aligned}
&e1 \leftarrow e2 \vee e3.\\
&e2([T_1,T_4]) \leftarrow a([T_1,T_2]) \otimes b([T_3,T_4]).\\
&e3([T_1,T_4]) \leftarrow c([T_1,T_2]) \otimes d([T_3,T_4]).
\end{aligned}
\tag{32}
$$

A graph, representing disjunction from rules (32), is shown in Figure 9 (where a disjunction node is depicted by $\vee$).

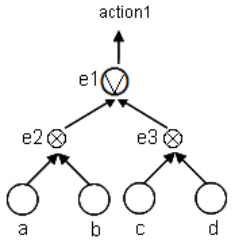

Fig. 9. Disjunction of events

Disjunction of a set of events is transformed into as many Horn rules as many disjuncts there are. Hence it is an easy operator to implement. In rules 33 the rule for $e1$ is transformed into two Horn rules.

$$
\begin{aligned}
&e1(1,[T_1,T_2]) \leftarrow e2([T_1,T_2]).\\
&e1(1,[T_1,T_2]) \leftarrow e3([T_1,T_2]).
\end{aligned}
\tag{33}
$$

An algorithm for detection of disjunction of events is trivial. Disjunction operator divides rules into disjuncts where each disjunct triggers the parent (complex) event. Hence we omit presentation of the algorithm here. However we have implemented the algorithm and used it to run our experimental tests (Section 6.7).

6.5 Negation and Event Processing

Negation in event processing is typically understood as absence of that event. In order to create a time interval in which we are interested to detect absence of an event, we define a negated event in scope of other complex events. In our case this is a sequence of events[17]. A rule (34) defines a pattern for $e1$ detected whenever an event a is followed by b, provided that c does not happen in between. This pattern is depicted in Figure 10.

$$
e1([T_1,T_4]) \leftarrow (a([T_1,T_2]) \otimes b([T_3,T_4])) \wedge c([T_5,T_6]). \tag{34}
$$

Rules (35)-(37) sketch how negation can be handled. We omit a complete set of rules due space limitation. Similarly as in Subsection 6.1, a sequence of a and b is

[17] Other operators such as concurrent conjunction are also applicable (on the other hand, disjunction is not).

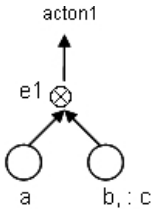

Fig. 10. Negation within a sequence of events

detected by rules (35)-(36). The difference is a rule (37). This rule is created to delete a goal that was inserted by a. Hence whenever c happens after a, it will reset the current state of detection of $e1$. In turn, $e1$ is detected after an occurrence of b, which follows a, with no c in between.

$$a(1,[T_1,T_2]) : -ins(goal(b([\_,\_]),a([T_1,T_2]),e1([\_,\_]))). \tag{35}$$

$$\begin{array}{l} b(1,[T_3,T_4]) : - goal(b([\_,\_]),a([T_1,T_2]),e1([\_,\_]))\otimes \\ del(goal(b([\_,\_]),a([T_1,T_2]),e1([\_,\_])))\otimes \\ e1([T_1,T_4]). \end{array} \tag{36}$$

$$c(1,[T_1,T_2]) : - del(goal(b([\_,\_]),a([T_1,T_2]),e1([\_,\_]))). \tag{37}$$

Algorithm 6.5 describes how to handle negation. After the procedure for dividing complex event rules into *binary event goals*, four different cases are distinguished, i.e., handling negation with respect to the semantics of *concurrent*, *classical*, *serial* conjunctions and disjunction[18]. In order to trigger e_1, concurrent, classical and serial conjunctions require (as a necessary condition) a to happen. The sufficient condition varies from a particular operator. In case of concurrent conjunction, $e1$ will be triggered whenever a occurs and b either does not happen at all, or it happens outside the time interval of a[19]. Similarly, use of classical conjunction with negation also requires a non occurrence of b on an interval. However note that the interval is now explicitally defined by user's pattern (i.e., $[T_1, T_6]$). Since classical conjunction does not demand the intervals overlap, there must still be a *finite interval* on which we monitor a non occurrence of b[20]. The same remark applies to sequential operator. However since sequential operator is not commutative, a non occurrence of b is monitored from the time when a has detected till the end of user specified interval. Disjunction ($e1^{[T_1,T4]} : -a \vee \neg b.$) is again trivial. $e1$ is triggered if either a occurs or b does not occur on $[T_1, T4]$, hence we omit here further consideration of disjunction with negation.

[18] In future, we will extend the list of operators, hence treatment of negation with those new operators will be defined too.

[19] Recall that in general case, for an event $a^{[T_1,T_2]}$, $T_1 \neq T_2$. Further on, we recall that concurrent conjunction ($a^{[Ti,T_i+m]}|b^{[T_j,T_j+n]}$) requires overlapping of the two intervals.

[20] Currently in our prototype implementation this interval is defined w.r.t other events monitored in the system. E.g., $[T_1, T_6]$ is defined in such a way that $T_1 = T_1(c)$ and $T_2 = T_2(d)$ where c and d are events defined in the event program.

Algorithm 6.4 Negation.
Divide complex event rules into binary event rules (e.g., $aop\neg b$) by introducing temporary event symbols (e.g., $e1, ...$);
For each event binary goal {
insert rules to generate goals:
$a :- while\_do(a, 1).$
switch w.r.t operator *op*:
case $(e1 :- a|\neg b.)$: {
$a^{[T_1,T_4]}(1) :- \{\neg(goal(\_, b^{[T_2,T_3]}, \_)) \vee [goal(\_, b^{[T_2,T_3]}, \_) \otimes (T_1 < T_2 \vee T_3 < T_4)]\} \otimes$
$e1^{[T_1,T4]}.$
break;
case $(e1[T_1, T_6] :- a \wedge \neg b.)$: {
$a^{[T_1,T_4]}(1) :- \{\neg(goal(\_, b^{[T_2,T_3]}, \_)) \vee [goal(\_, b^{[T_2,T_3]}, \_) \otimes (T_1 < T_2 \vee T_3 < T_4)]\} \otimes$
$e1^{[T_1,T4]}.$
break;
case $(e1[T_1, T_5] :- a \otimes \neg b.)$: {
$a^{[T_1,T_4]}(1) :- \{\neg(goal(\_, b^{[T_2,T_3]}, \_)) \vee [goal(\_, b^{[T_2,T_3]}, \_) \otimes (T_1 < T_2 \vee T_3 < T_4)]\} \otimes$
$e1^{[T_1,T4]}.$
break;
insert check rules:
$b :- while\_do(b, 1).$
$b^{[T_1,T_2]}(1) :- ins(goal(\_, b^{[T_1,T_2]}, \_)).$
}

Similar simple and clear transformations can be written for the other event operators. Due to space constraints, in the remaining part of this section, we briefly describe the other operators for complex events defined by Definition 5, but leave out the actual transformation algorithms.

6.6 N-times and *-times Sequences of Events

An N-times sequence of events is detected when an event is fired continuously N times. For instance, the rule (38) defines $e1$ as a conjunction of three complex events, i.e., event a was fired, b was fired 10 times, followed by event c (for simplicity we present these events without times while times are easy to introduce if necessary). We note that the formula $b * N$ can be represented as: $b \otimes b \otimes ... \otimes b$ (N times), but this cannot be done in the static transformation if N is a variable, which gets instantiated during evaluation from the arguments of the events (e.g., $a(N) \otimes b * N$).

$$e1 \leftarrow a \otimes b * 10 \otimes c. \tag{38}$$

The event equation from (38) is transformed into the CTR formulas (39)-(44), where $e2$ and $e3$ are temporary events introduced by binarization of rules and N-times operator. The rules with the head $b/1$ trigger two procedures: a procedure to enter in the event e3 and a procedure to execute the loop for the event $e2$ N times. A graph representing N-times from the rules (39)-(44), is shown in Figure 11 (where the N-times construct is depicted by the loop for the event e2).

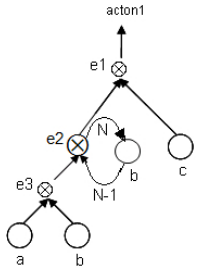

Fig. 11. N-times sequence of events

$$a :- while\_do(a, 1).$$
$$a(1) :- ins(goal(b, a, e3)). \tag{39}$$

$$b :- while\_do(b, 1).$$
$$b(1) :- goal(b, a, e3) \otimes del(goal(b, a, e3)) \otimes e3. \tag{40}$$
$$b(2) :- goal(e3, e2, N) \otimes del(goal(e3, e2, N)) \otimes e2(N).$$

$$c :- while\_do(c, 1).$$
$$c(1) :- goal(e2, c, e1) \otimes del(goal(e2, c, e1)) \otimes e1. \tag{41}$$

$$e1 :- while\_do(e1, 1).$$
$$e1(1) :- action1. \tag{42}$$

$$e2 :- while\_do(e2, 1, N).$$
$$e2(1, N) :- N = 0 \otimes ins(goal(e2, c, e1)).$$
$$e2(1, N) :- N > 0 \otimes ins(goal(e3, e2, N - 1)). \tag{43}$$

$$e3 :- while\_do(e3, 1).$$
$$e3(1) :- N = 10, N1 is N - 1, ins(goal(e3, e2, N1)). \tag{44}$$

The *-times operator is a simplified version of the N-times operator. The *-times sequences of events are detected when an event is fired at least once. For instance, the rule (45) defines $e1$ as a conjunction of three complex events, i.e., event a was fired, b was fired at least once, followed by event c. Such rules are very frequent in event stream processing applications, such as stock market feeds where a stock symbol can be traded an undefined number of times before an update of its price (possibly triggering an analysis of the new price).

$$e1 \leftarrow a \otimes b \uparrow * \otimes c. \tag{45}$$

The event equation from (45) is transformed into the CTR formulas (39)-(48), where $e2$ and $e3$ are temporary events introduced by binarization of rules and the *-times operator. The rules with the head $b/1$ trigger two procedures: a procedure to enter in the event e3 and a procedure to execute the loop for the event $e2$. The event b can be executed an undefined number of times before the execution of c takes place. A graph representing these rules is shown in Figure 12 (where the *-times construct is depicted by the loop for the event e2).

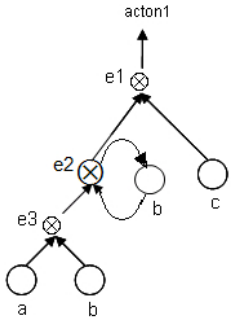

Fig. 12. *-Times sequence of events

$$b(2) : -goal(e2, c, e1) \otimes del(goal(e2, c, e1)) \otimes e2. \tag{46}$$

$$e2(1) : -ins(goal(e2, c, e1)). \tag{47}$$

$$e3(1) : -ins(goal(e3, b, e2)). \tag{48}$$

6.7 Implementation

In this subsection we present first preliminary results of our prototype implementation. We have implemented algorithms that transform user defined event pattern rules into rules suitable for event-driven pattern detection, and execute events in XSB Prolog. All the test cases were run on a workstation with Pentium dual-core processor 2,4GHz CPU and 3GB memory running on Ubuntu Linux and XSB Prolog version 3.1. In our experiments we have used different operators to generate a few events in a block of complex patterns. The first experiment consists of six atomic events (i.e., $e1_1$-$e1_6$) and 4 complex events (i.e., $ce1_1$ - $ce1_4$). Further, we have been multiplying the number of event blocks to assess the throughput of input (atomic) events that can be handled (i.e., used for detection of complex patterns) by our prototype. The sequences are of depth between two and one million events, see experiment rules (49) below.

$$\begin{array}{l} ce1_1 : -e1_1 \otimes e1_2. \\ ce1_2 : -e1_2 \otimes e1_3 \otimes e1_4. \\ ce1_3 : -(e1_3 \otimes e1_5)|(e1_4 \otimes e1_6). \\ ce1_4 : -(e1_2 \vee e1_3) \wedge (ce1_2|\neg e1_5). \end{array} \tag{49}$$

Results. Figure 13 shows the experimental results obtained with the prototype. The test is repeated an increasing number of times. The maximum throughput that we got is in a range between 1100 and 1200 events/second. We tested for sizes of 10,000 blocks (4438ms), 50,000 (21360ms) and 500,000 (237950ms).

The memory consumption during the pattern matching process for the first experiment is very low and constant. Irrelevant events are removed from memory as soon

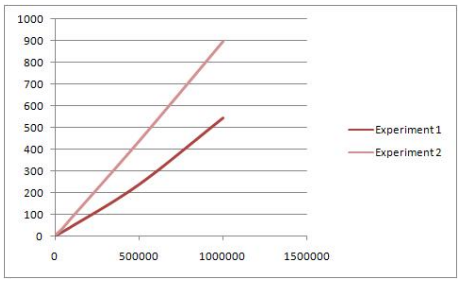

Fig. 13. The event throughput for our experiments

as they are "consumed" (i.e., propagated upward or triggered). In our real experiments with stock data, the memory consumption was a bit higher, due to necessity to keep more intermediate events alive. Still the consumption was insignificant with respect to the current RAM memories (a few percent).

The second experiment consists of a very deep complex event detection. The event $startExperiment2$ triggers a very long chain of events: $startExperiment2$ starts $ce2_1$, further $ce2_1$ starts $ce2_2$, $ce2_2$ starts $ce2_3$ and so on.

$$\begin{array}{l} eventTrigger(startExperiment2). \\ ce2_1 : -startExperiment2. \\ ce2_2 : -ce2_1. \\ ce2_3 : -ce2_2. \\ ... \\ ce2_n : -ce2_{n-1}. \end{array} \tag{50}$$

The second experiment consists of compiling the real end-of-day market stock values as event triggers for the IBM company in the last 40 years (10K stock ticks). Each tuple has: the date, the opening and closing price, and the low and the high for the day. A composed event is triggered when the opening price is higher then the closing price (51). The execution time was 9.408sec, confirms our estimate of approx. 1000 events per second.

$$ce3(Date) : -stock('IBM', Volume, Date, Open, Close, Low, High) * Open > Close. \tag{51}$$

In this subsection we have provided measurement results. Even though there is a lot of room for improvements and optimizations, preliminary results show that logic-based event processing has capability to handle significant amount of events in reasonable time. Taking account its strength (i.e., robust inference capability), it promises a powerful approach for combining *deductive* and *temporal* capabilities in a unified framework, yet with good run-time characteristics.

7 Logic-based ECA Rules

Event-Condition-Action rules are used in event-driven information systems where reactive behavior is required, i.e., systems capable to detect events and respond to them

automatically [16]. The context, in which events are triggered, is also taken into account (by handling the condition part). The general form of ECA rules is: "ON *Event* IF *Condition* DO *Action*".

In this section we review the role of basic elements of an ECA rule (i.e., event, condition, action), putting them in a logical framework, and implementing them with $\mathcal{CTR}$. Our goal is to implement *event processing* in a logical framework extended with *reactions* on those events, also implemented with a logic. Such an approach give us a unique capability to reason over the all reactive system, keeping the control of behavioral characteristics of the system with the logic.

7.1 Event and Condition

In general, reactive systems are recursive systems, where event is a central notion for driving the execution. Hence the role of events is, first, to identify situations in which the system is supposed to react; second, to start the execution of ECA programs in an appropriate moment. The implementation of the event part corresponds to the implementation of the event pattern.

The context, in which an ECA rule fires, is described by the condition part. It determines whether the rule (triggered by a certain event pattern) will execute or not. The condition part is usually represented as a query. In our case, the context (and hence the condition of an ECA rule) may be more complex. $\mathcal{CTR}$ has rich capabilities to query a database in every particular state during the execution [5]. Therefore, context, in our framework, is represented by the *state* of a reactive system. It can contain any number of Boolean conditions; or a database containing all facts that are true in that state. Moreover a context can be represented with implicit information stated in form of rules. This in turn allows an inference engine to reason about the current context, prior to initiating an action for execution or raising an event. Process of discovering (reasoning about) the current context is necessary since reaction (i.e., another event pattern or an action) is valid only in a certain context. One particular event pattern can be interpreted in different ways, and hence can trigger different actions or events, when occurred in different situation. We say that one event in a particular context consists one particular *situation*.
Definition (Context). The context is defined by a $\mathcal{CTR}$ rule, where the rule head is the context name, and the rule body is the context definition ■

In the following example we show a complex context which is dependent on a few other sub-conditions and tests.

$ctx(T, T_1, T_2) \leftarrow ctx_1(T_1) \wedge (cond_2(T_2) \vee cond_3(T_2)) \wedge$
$\neg cond_4(T, T_1, T_2) \wedge (T_2 - T_1 < 10sec.$
$ctx_3(T) \leftarrow ctx_1(T) \wedge ctx_5(T).$

Processing of complex contexts may affect performance of event processing engines. Many of "state of the art" event pattern rule languages cannot express complex contexts (providing only the condition part for querying the data). The approach based

on $\mathcal{CTR}$[21] supports not only specification of complex constraints, but also their evaluation (since it is based on logic) and reasoning about event patterns in particular contexts (i.e., situations).

7.2 Action

The action part changes the state of a system. While events are triggered as a consequence of state changes, the actual state changes are caused by actions. Hence the reactive behavior of ECA systems is realized through the execution of actions. Typical examples of actions are: updating persistent data, calling a web service, triggering new events, committing a database transaction, or the rule base modification.

In general case, the purpose of the action part is to change the state of the system. An example of the state change is a single update in the knowledgebase. However atomic actions, such as data update, are too limiting in practise. More often we need to combine atomic actions into *complex actions*. We extend the standard ECA framework with deductive capabilities such that the action part can be formally described with $\mathcal{CTR}$. First, our extension is motivated by the aim to integrate *active behavior* (from ECA rules) with *deductive capabilities* (from $\mathcal{CTR}$). Second, the extended framework integrate *reactive* and *continuous* behavior appropriately.

In the following, we give legal possibilities for creating a complex action from atomic ones.

Definition (Action). An action is a formula of the following form:

- an atomic action;
- $(action_1 \vee action_2 \vee ... \vee action_n)$, where $n \geq 0$ and each $action_i$ is an action (Disjunctive composition);
- $(action_1 \otimes action_2 \otimes ... \otimes action_n)$, where $n \geq 0$ and each $action_i$ is an action (Sequential composition);
- $(action_1 \mid action_2 \mid ... \mid action_n)$, where $n \geq 0$ and each $action_i$ is an action (Concurrent composition);
- $\neg action$, where $action$ is an action (negation).
- $\odot action$, where $action$ is an action (isolation).

A rule is a formula of the form $actionA \leftarrow actionB$, where $actionA$ is an atomic action, and $actionB$ is either an atomic or a complex action ■

A rule may be seen as an action procedure, where the rule head is a complex action name, and the rule body is the action definition. Likewise events, actions are not just propositions, but contain data terms. Each action $a(X_1, X_2, ..., X_n)$ may be of arity n, $n \geq 0$, where $X_1, X_2, ..., X_n$ is a list of variables or constants, representing parameters of the action procedure.

Utilising $\mathcal{CTR}$ operators, complex actions may create complex processes that, at the end, may form a workflow [3]. As we can see, from Section *CTR Overview* and

[21] Datalog¬ is a subset of $\mathcal{CTR}$, and Datalog¬ like rules (used to express the context) are called database rules in $\mathcal{CTR}$.

Definition *Action*, actions may run in parallel possibly having non-serializable access to shared resources, or for instance, they can communicate and synchronise themselves.

As mentioned in Section *CTR Overview*, $\mathcal{CTR}$ has a notion of *execution paths*. Execution paths show the way complex actions are executed, and record their execution history. Formally, an execution path of a complex action is represented as a finite sequence of pairs: $S_1S_2, S_3S_4, ..., S_{n-1}S_n$, where each state-change (i.e., S_iS_{i+1}) represents a period of an atomic action execution. For instance, imagine an event e_1 has occurred, and caused a corresponding complex action a_1 to start executing. The event e_1 has occurred when the knowledgebase was in the state S_1, and after the execution of the action a_1, the system will be brought to the state S_n. Since the a_1 is a complex action, the system will go through a set of states $S_1S_2, S_3S_4, ..., S_{n-1}S_n$. Every state transition, S_iS_{i+1}, corresponds to execution of an atomic action. Now suppose that during the execution of a_1, an event e_2 has occurred. The event e_2 has caused an action a_2 to happen, which is a simple action and hence will be completed before a_1. Note that a_2 will change the state of the whole system, while a_1 is still executing. Thanks to an *interleaving semantics* of $\mathcal{CTR}$, after the execution of both actions, the whole ECA system will still remain in the consistent state. The interleaving semantics assumes a single execution path although a number of concurrent complex actions may be running at the same time. Every complex action consists of a sequence of sub-actions, where each sub-action changes a state of the system. However by interleaving these sequences, we obtain a new sequence of state changes, which is an execution path.

However some complex actions need to be executed continuously (i.e., without interruption by other sub-actions, or suspension). In this situation we use a *modality of isolation*. For example, consider a complex *actionA* which consists of three sub-actions:

$$\textit{actionA} \leftarrow \odot\,[taskA_1 \otimes taskA_2 \otimes taskA_3]$$

An execution path of the above action is a single pair of states, S_1, S_2. Although *actionA* is a complex action, it has been modeled to change the state only from S_1 to S_2 (not as a sequence of changes: S_1S_2, S_2S_3, S_3S_4). Therefore if some other actions are executing at the same time, their execution paths will not be interleaved with the one of *actionA*. Therefore assuming that *actionA* started to execute first, the other actions will be committed after the *actionA*.

The following example from [4], demonstrates another important feature of $\mathcal{CTR}$, that is *synchronization*.

Example (Synchronization).
actionA $\leftarrow$ *actionB* | *actionC*.
actionB $\leftarrow taskB_1 \otimes ins.startC_2 \otimes taskB_2 \otimes startB_3 \otimes taskB_3$.
actionC $\leftarrow taskC_1 \otimes startC_2 \otimes taskC_2 \otimes ins.startB_3 \otimes taskC_3$.

The first rule defines a complex *actionA*, which consists of two sub-actions: *actionB* and *actionC*. The two sub-actions execute concurrently, but not independently. In particular, each action performs three tasks, where $taskC_2$ (from *actionC*) cannot start until $taskB_1$ (from *actionB*) is finished. Similarly, $taskB_3$ cannot start until $taskC_2$ is

completed (i.e., specified with $ins.startB_3$ from $actionC$, and an atom $startB_3$ from $actionB$). Therefore we see that *actionB* communicates with *actionC*. Moreover the two sub-actions, *actionB* and *actionC*, are synchronized between themselves.

In case we want to synchronize the two actions with some (external) events, we may replace constructs for the synchronization (i.e., $ins.startC_2$ and $ins.startB_3$) with some events $startC_2$, $startB_3$, and hence achieve tighter integration between events and actions. Moreover this integration has been achieved at the logical level, which allows reasoning about actions and events. For instance a pattern such as "notify me if an *actionA* happened before an *eventE*, and the *eventE* happened before some *actionB*" would be easy extractable by the reasoner.

7.3 Completing and Executing ECA Rules

Once the consisting elements of an ECA rule are defined, we can glue them together demonstrating how ECA rules are implemented in $\mathcal{CTR}$. Let us observe first the condition-action (C-A) pattern of an ECA rule. To implement this pattern one can utilize an implementation of **if-then-else** statement in declarative way, as shown in [6]:

$$\begin{array}{l} if\_a\_b\_c \leftarrow (\Diamond a) \otimes b \\ if\_a\_b\_c \leftarrow (\Box \neg a) \otimes c \end{array} \tag{52}$$

The above rules executes b if a condition a is satisfied, else c will execute[22]. Further on, only **if-then** statement can be represented with the following:

$$\begin{array}{l} if\_a\_b\_c \leftarrow (\Diamond a) \otimes b \\ if\_a\_b\_c \leftarrow \Box \neg a \end{array} \tag{53}$$

In the same way we can write a C-A rule:

$$\begin{array}{l} c\_a\_rule \leftarrow (\Diamond cond) \otimes act \\ c\_a\_rule \leftarrow \Box \neg cond \end{array} \tag{54}$$

We shell write **if cond then act** as an abbreviation for the proposition c_a_rule. So the complete ECA rule is then:

$$\begin{array}{l} event(X) \leftarrow c\_a\_rule \\ c\_a\_rule \leftarrow \textbf{if } cond \textbf{ then } act \end{array} \tag{55}$$

If there are n actions to be triggered on $event(X)$ we write:

$$event(X) \leftarrow c\_\_a\_rule_1 | c\_\_a\_rule_2 | ... | c\_\_a\_rule_n \tag{56}$$

providing for each action a condition:

[22] $if\_a\_b\_c$ is a name of this particular **if-then-else** statement. The name should not be used as a head of any other rule.

$$
\begin{array}{l}
c\_a\_rule_1 \leftarrow \textbf{if } cond_1 \textbf{ then } act_1 \\
c\_a\_rule_2 \leftarrow \textbf{if } cond_2 \textbf{ then } act_2 \\
... \\
c\_a\_rule_n \leftarrow \textbf{if } cond_n \textbf{ then } act_n
\end{array} \tag{57}
$$

The rule in (56) executes n actions in parallel. This means that there is no a preferred order over the execution of actions. However in some situation it is desirable to trigger actions in a certain order (by implementing a special policy of an application). For example, actions need to execute in reverse order (from the rule in (56)). In that situation, $\mathcal{CTR}$ offers an easy way to implement such a conflict resolution strategy:

$$
event(X) \leftarrow c\_a\_rule_n \otimes c\_\_a\_rule_{n-1} \otimes ... \otimes c\_\_a\_rule_1 \tag{58}
$$

Rules from (55) and (58) demonstrate how ECA rules can be completely implemented in a declarative way, allowing for a unifying model theory for the framework (easy to extend with other logical formalisms, such as: Hilog, Defeasible logics, etc). ECA rules are considered as an appropriate form of *reactive rules*. However their use in practice very often can be unpredictable, with respect to their intended semantics [10]. In general case, execution of an event may trigger other events, and these events may trigger even more events. There is neither guarantee that, such a chain of events will stop, nor that states (through which a reactive system passes) are valid. We see semantics as a means to establish some sort of a *consistency check* mechanism in an ECA reactive system. The purpose of this mechanism is to control state-changing actions, keeping the system always in the consistent state. By executing a set of complex ECA rules, a reactive system implemented with $\mathcal{CTR}$ changes its states. In this transition, every state in which the system enters, needs to be a legal state (with respect to the ECA rules and the semantics provided by $\mathcal{CTR}$). However if the inference engine, searching for a possible execution path, enters to an illegal state (w.r.t the semantics of given rules), such a state-transition will be rolled back. In this sense ECA rules implemented in a logical framework, allow for an easier control over the entire event-driven system during the run-time.

8 Event Pattern Constraints

An event pattern constraint expresses a condition which must be satisfied by the events observed in an event-driven system. According to [11], a constraint is used to test certain patterns of events that *never happen* in the system. For example, with a constraint we express a policy that an order from a customer must never be first confirmed and later denied. We can take a broader definition of a constraint. Apart from expressing events that must never happen, constraints can also guarantee which events *must happen*. So constraints serve to force an event-driven system to behave in a way specified by constraints (e.g., a particular event pattern must happen, otherwise the current execution is not be possible).

Implementation of a powerful event constraint algebra with $\mathcal{CTR}$ have already been specified in [17, 18]. Therefore we will not go into details of $\mathcal{CTR}$ event constraints

here. Rather for the sake of completeness we show that $\mathcal{CTR}$ is a unifying logic formalism suitable for CEP, including event constraint handling too. Example *Constraints* illustrates a couple of event constraints that can be expressed in $\mathcal{CTR}$. For the further details, a reader is referred to [17, 18]. Since the truth values of $\mathcal{CTR}$ formulas are defined on paths (see Section *CTR Overview*), let us denote an event constraint e (which needs to occur somewhere on an execution $path$) with $path \otimes e \otimes path$ (abbreviated ∇e). Then the following formulas are all legal event constraints implemented in $\mathcal{CTR}$.

Example (Constraints).
$\nabla e_1 \vee \neg\nabla e_2$ - If event pattern e_1 occurs, then e_2 must also occur (before or after e_1);
$\neg\nabla e_1 \vee (\nabla e_1 \otimes \nabla e_2$ - If event pattern e_1 occurs, then e_2 must occur after e_1;
$\neg\nabla e_1 \vee \odot(e_1 \otimes e_2)$ - If event pattern e_1 occurs, then e_2 must occur right after e_1 with no event in-between.

9 Conclusions and Future Work

We propose a novel approach for Complex Event Processing based on an encoding in Transaction Logic programs. The approach clearly extends capabilities of Active Databases with declarative semantics and the power of rule-based reasoning. Further on, Active Databases usually combine two or more formalisms: a high-level language for procedural programming (e.g., Java, C, Python), and a formalism for the complex action specification and execution (e.g., process algebra as in [2], SQL, XPath/XQuery, Datalog, SPARQL, F-Logic, deterministic automatas in [1] and [12] etc.). $\mathcal{CTR}$ does not make a sharp distinction between declarative and procedural programming. Therefore, the proposed framework provides a seamless integration of these two programming styles. The encoding in $\mathcal{CTR}$ allows specification of more complex event patterns, contexts (conditions), actions, and event constraints, and is easily combined with other logical formalisms, such as: prioritized and defeasible logics, Hilog and F-logic. We believe this approach is also more pragmatic from the implementation and optimization point of view. For the next steps we will continue to study advantages and drawbacks of Complex Event Processing implemented in a logical framework and work on the implementation of transformation algorithms for new event operations in the $\mathcal{CTR}$-based event processing (e.g., the isolation operator and aggregate predicates for complex events).

10 Acknowledgment

We thank Ahmed Khalil Hafsi and Jia Ding for their help in implementation and testing of our prototype.

References

1. J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD Conference*, pages 147–160, 2008.

2. E. Behrends, O. Fritzen, W. May, and F. Schenk. Combining eca rules with process algebras for the semantic web. In *RuleML*, 2006.
3. A. J. Bonner. Workflow, transactions and datalog. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 1999.
4. A. J. Bonner and M. Kifer. Transaction logic programming (or, a logic of procedural and declarative knowledge. In *Technical Report CSRI-270*, 1995.
5. A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996.
6. A. J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In *DBLP-4: Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*. Springer-Verlag, 1994.
7. F. Bry and M. Eckert. Rule-based composite event queries: The language xchangeeq and its semantics. In *RR*. Springer, 2007.
8. F. Bry and M. Eckert. Towards formal foundations of event queries and rules. In *Second Int. Workshop on Event-Driven Architecture, Processing and Systems EDA-PS*, 2007.
9. P. Haley. Data-driven backward chaining. In *International Joint Conferences on Artificial Intelligence*. Milan, Italy, 1987.
10. M. Kifer, A. Bernstein, and P. Lewis. *Database Systems - An Application-Oriented Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.
11. D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.
12. A. Majumder, R. Rastogi, and S. Vanama. Scalable regular expression matching on data streams. In *SIGMOD Conference*, pages 161–172, 2008.
13. R. Milner. Calculi for synchrony and asynchrony. In *Theor. Comput. Sci.*, 1983.
14. A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rules language for complex event processing. In *International Workshop on Event Drive Architecture for Complex Event Process*. ACM, 2007.
15. N. W. Paton and O. Díaz. Active database systems. In *ACM Comput. Surv.* ACM, 1999.
16. N. W. Paton, F. Schneider, and D. Gries. *Active Rules in Database Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.
17. D. Roman and M. Kifer. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowmentg, 2007.
18. D. Roman and M. Kifer. Semantic web service choreography: Contracting and enactment. In *International Semantic Web Conference*. Springer, 2008.